R Small Group: Class 5

Amy Pomeroy & Dayne Filer

March 7, 2018

Using this document

- Code blocks and R code have a grey background (note, code nested in the text is not highlighted in the pdf version of this document but is a different font).
- # indicates a comment, and anything after a comment will not be evaluated in R
- The comments beginning with ## under the code in the grey code boxes are the output from the code directly above; any comments added by us will start with a single #
- While you can copy and paste code into R, you will learn faster if you type out the commands yourself.
- Read through the document after class. This is meant to be a reference, and ideally, you should be able to understand every line of code. If there is something you do not understand please email us with questions or ask in the following class (you're probably not the only one with the same question!).

Class 5 expectations

1. Write and run a basic function in R
2. Understand function environments and how functions find things
3. Understand the "do not repeat yourself" (DRY) principle

Basic functions

So far we've used a lot of functions that already exist in R. Now you will learn how to write your own functions. User-written functions take the following structure:

```
myfunction <- function(arg1, arg2, ...) {

  do some stuff
  return(object)

}
```

Make a simple function that just returns the arument that it is given.

```
first_function <- function (x) {

  return(x)

}
```

Now if you pass a value to `first_function` it should return that value:

```
first_function(9)
## [1] 9
```

A function that only returns the given value is not useful, but it does illustrate how you can add functions to your environment. Now make a function that squares a value and then adds one to it.

```
second_function <- function(x) {

  ans <- x^2 + 1
  return(ans)

}
second_function(9)
## [1] 82
```

The function environment

An environment is a place to store variables. As we discussed in class 1, when you make assignments in R, they are generally added as entries to the global environment.

Functions are evaluated in their own environments. When a function is called a new environment is created. This new environment is called the evaluation environment. Functions also have an enclosing environment, which is the environment where the function was defined. For a functions defined in the workspace the enclosing environment is the global envrionment.

When a function is evaluated, R looks through a series of environments for variables called. It first searches the evaluation environment and then the enclosing environment. This means that a global variable can be referenced inside a function. This principle is shown below by `third_function` which sums the argument `x` with the variable `a` from the global environment.

```
a <- 9
third_function <- function(x) { x + a }
third_function(11)
## [1] 20
```

The evaluation environment is populated with local variables as the evaluation of the function procedes. Once the function completes running, the evaluation environment is destroyed. Look at `second_function` for example. Within the function the variable `ans` is assigned. However, when you evaluate the function the `ans` variable is not available the global environment because it was assigned in the evaluation environment, and the evaluation environment was destroyed after the function completed.

```
second_function(9)
## [1] 82
ls()
## [1] "a"               "first_function"  "second_function" "third_function"
```

As you can see, listing objects in the global environment only returns the global variable `a` assigned previously and the three functions that you have defined so far.

DRY principle

The don't repeat yourself (DRY) principle states that

> every piece of knowledge must have a single, unambigous, authorative representation within a system - Dave Thomas and Andy Hunt.

There is a lot to think about when using the DRY principle, like not repeating variable names throughout your code. The DRY principle is particularly useful when thinking about functional programming. If you find

yourself writing the same code more than once, or copying and pasting, consider writing a function. Think back to the primer list discussed in last class, and consider the following code to generate two sets of fruit:

```
library(stringi)
lengths1 <- sample(15:35, 25,replace=TRUE)
primers1 <- list(primer = stri_rand_strings(25, lengths1, '[ACTG]'),
                 length = lengths1)
lengths2 <- sample(20:40, 25,replace=TRUE)
primers2 <- list(primer = stri_rand_strings(25, lengths1, '[ACTG]'),
                 length = lengths1)
```

Now imagine you have to create 100 primer lists. After you fininsh you realize you need to change the list of possible primer lengths. If you wrote code like above you would have to edit each line. However, you may see how the code above could be functionalized. Consider what may be variable in the code above, and create a function parameter for each variable part of the code. You may wish to change the minimum primer length, the maximum primer length, the number of primers in each list, and even the possible bases.

```
## Create a funciton to primer lists.
genPrimer <- function(minlen, maxlen, nprimers, bases = '[ACTG]') {

  lengths <- lengths1 <- sample(minlen:maxlen, nprimers,replace=TRUE)
  res <- list(primer = stri_rand_strings(nprimers, lengths, bases),
              length = lengths)
  return(res)

}
primers1 <- genPrimer(minlen = 15, maxlen = 35, nprimers = 25)
primers2 <- genPrimer(minlen = 20, maxlen = 40, nprimers = 25)
```

Notice two things: (1) now any change needed in the fruit generation only requires the code to change in one place – greatly reducing the amount of work, and more importantly, the change for error; (2) notice how 'bases' is defined in the function with a default value. You can probably imagine how many functions benefit from default parameter values. For the `genPrimer` function above you will most often only generate sequences with nucelotides found in DNA, so it makes sense to provide a default value for the bases. Almost all of the functions you will use in R run with default values that you may or may not be aware of.

Similarly, you can functionalize the molecular weight and Tm calculators:

```
library(stringr) #This library is required for the function that calculates Tm
primer_analysis <- function(primers) {

  # create results list
  res <- list (primer = character(), mw = numeric(),
               Tm = numeric(), length = numeric())

  ## For-loop iterating through the 1:(number of fruit)
  for (p in 1:length(primers$primer)) {

    sequence <- primers$primer[p] ## Get the fruit type from the list
    len <- primers$length[p] ## Get the fruit width from the list

    ### Get molecular weight
    pmass <- 0 #set mass of primer equal to zero
    for (j in 1:len){
```

```
      l <- substr(sequence,j,j) #get jth nucleic acid of ith primer
      if (l=="C"){
        m <- 467.2
      } else if (l=="A"){
        m <- 491.2
      } else if (l=="G"){
        m <- 507.2
      } else if (l=="T"){
        m <- 482.2
      } #get mass of jth nucleic acid of ith primer
      pmass <- pmass+m #add mass of jth nucleic acid to the mass of the primer
    }

    ### Get Tm
    nA <- sum(str_count(sequence,"A")) #number of A's
    nC <- sum(str_count(sequence,"C")) #number of C's
    nG <- sum(str_count(sequence,"G")) #number of G's
    nT <- sum(str_count(sequence,"T")) #number of T's
    if (len<=13){
      Tm <- (nA+nT)*2 + (nG+nC)*4
    }
    else{
      Tm <- 64.9 +41*(nG+nC-16.4)/(nA+nT+nG+nC)
    }

    # assign values to the results matrix
    res$primer[p] <- sequence
    res$mw[p] <- pmass
    res$Tm[p] <- Tm
    res$length[p] <- len

    }

    return(res)
}
```

Now you can use the `primer_analysis` function to the `primers1` and `primers2` datasets you generated.

```
results1 <- primer_analysis(primers1)
results2 <- primer_analysis(primers2)
```

Function control statements

So far we have only discussed one aspect of function control statments: the `return` function. The `return` function can actually go anywhere in the function, and a function can have multiple `return` functions. For example consider a function that returns "big number" for numbers greater than or equal to 100 and "small number" for numbers under less than 100:

```
xmpl_func <- function(x) {

  if (x >= 100) {
    return("big number")
```

```
  }

  return("small number")

}
xmpl_func(1)
## [1] "small number"
xmpl_func(1e10)
## [1] "big number"
xmpl_func("see what happens")
## [1] "big number"
```

When a function encounters the `return` function the returns the value and halts any execution. The other two functions worth knowing are: `stop` and `warning`. The `stop` and `warning` functions allow you to do tests within the function and exit the function (`stop`) if necessary, or issue a warning (`warning`).

In the `xmpl_func("see what happens")` example, the function returned `TRUE`. (Recall from class 1 that both would be converted to character in the if statement, which is why the code did not error.) However, you may wish to make sure the input to `xmpl_func` is numeric and has a length of 1. Modify `xmpl_func` to test the input.

```
xmpl_func <- function(x) {

  ## Test the length of x
  if (length(x) > 1) stop("'x' must be of length 1.")

  ## Test the class of x
  if (!is.numeric(x)) stop("'x' must be numeric.")

  if (x >= 100) {
    return("big number")
  }

  return("small number")

}
xmpl_func("see what happens")
## Error in xmpl_func("see what happens"): 'x' must be numeric.
xmpl_func(1:10)
## Error in xmpl_func(1:10): 'x' must be of length 1.
```

Finally, you may wish to issue a warning if the input can be converted to numeric instead of an error:

```
xmpl_func <- function(x) {

  ## Test the length of x
  if (length(x) > 1) stop("'x' must be of length 1.")

  if (is.logical(x)) {
    x <- as.numeric(x)
    warning("'x' was converted from logical to numeric.")
  }
```

```
  if (!is.numeric(x)) stop("'x' could not be converted to numeric.")

  if (x >= 100) {
    return("big number")
  }

  return("small number")

}
xmpl_func("see what happens")
## Error in xmpl_func("see what happens"): 'x' could not be converted to numeric.
xmpl_func(TRUE)
## Warning in xmpl_func(TRUE): 'x' was converted from logical to numeric.
## [1] "small number"
```

Notice the function completed when the input was logical. The `warning` function does not halt execution, but does print a warning message for the user.

Small Group Exercises

These exercises are to help you solidify and expand on the information given above. We intentionally added some concepts that were not covered above, and hope that you will take a few minutes to think through what is happening and how R is interpreting the code.

1. Write a function that solves the quadratic formula. Recall, given:

$$ax^2 + bx + c = 0$$

   The quadratic equation is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Use it to solve the following equations.

$$x^2 + x - 4 = 0$$
$$x^2 - 3x - 4 = 0$$
$$6x^2 + 11x - 35 = 0$$

2. Add the necessary checks for input length and class to the quadratic formula function from question 1.